Physical Simulation for Animation and Visual Effects: Parallelization and Characterization for Chip Multiprocessors

Christopher J. Hughes†, Radek Grzeszczuk‡*, Eftychios Sifakis§*,
Daehyun Kim†, Sanjeev Kumar†, Andrew P. Selle§*, Jatin Chhugani†,
Matthew Holliman†, Yen-Kuang Chen†
†Microprocessor Technology Labs, Intel ‡Nokia Labs §Stanford University
christopher.j.hughes@intel.com

ABSTRACT

We explore the emerging application area of physics-based simulation for computer animation and visual special effects. In particular, we examine its parallelization potential and characterize its behavior on a chip multiprocessor (CMP). Applications in this domain model and simulate natural phenomena, and often direct visual components of motion pictures. We study a set of three workloads that exemplify the span and complexity of physical simulation applications used in a production environment: fluid dynamics, facial animation, and cloth simulation. They are computationally demanding, requiring from a few seconds to several minutes to simulate a single frame; therefore, they can benefit greatly from the acceleration possible with large scale CMPs.

Starting with serial versions of these applications, we parallelize code accounting for at least 96% of the serial execution time, targeting a large number of threads. We then study the most expensive modules using a simulated 64-core CMP.

For the code in key modules, we achieve parallel scaling of 45x, 50x, and 30x for fluid, face, and cloth simulations, respectively. The modules have a spectrum of task granularity and locking behavior, and all but one are dominated by loop-level parallelism. Many modules operate on streams of data. In some cases, modules iterate over their data, leading to significant temporal locality. This streaming behavior leads to very high on-die and main memory bandwidth requirements. Finally, most modules have little inter-thread communication since they are data-parallel, but a few require heavy communication between data-parallel operations.

Categories and Subject Descriptors: C.1.4 [Parallel Architectures]; J.2 [Physical Sciences and Engineering]: Physics

General Terms: Performance, Measurement.

Keywords: CMP, physical simulation, characterization, parallelization.

*Work done while at Intel.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

1. INTRODUCTION

In designing future generations of microprocessors, it is critical that we consider the requirements of future workloads. One key emerging workload domain is computer animation through physical simulation. Physical simulation applications model complex natural phenomena, such as ocean waves crashing on a shore, or a flag waving in the wind by means of numerical simulation of physical laws. Physical simulation can be used in a variety of settings such as weather prediction, movie special effects, and computer games. Modeling different natural phenomena requires a diverse set of techniques, algorithms, and data structures, making physical simulation both complex and general.

This paper focuses on physical simulation intended for visualization (e.g., the output can be rendered for inclusion in feature films). The goal of these applications is to recreate the visual experience of a human observing a natural phenomenon. This leads to an emphasis on visual plausibility and simulation efficiency rather than 100% accuracy in modeling all intermediate phenomena. Therefore, for the majority of phenomena simulated, the applications target the scale and level of detail that is perceivable by a human observer. This typically excludes physical phenomena happening at either the microscopic level (e.g., molecular dynamics or cell evolution) or beyond our everyday experience macroscopic level (e.g., planetary scale phenomena).[1]

The applications we study are based on the PhysBAM package from Stanford [9], which is used by several special effects and film production companies, including Industrial Light and Magic® and Pixar®. This class of physical simulation workloads has received little attention from the architecture community outside of groups working on high performance computing. It is extremely demanding in terms of computation and memory requirements. This makes the workloads a challenging target for future as well as current architectures. Furthermore, these workloads are likely precursors of future computer games and other mass-market applications.

It is highly desirable to accelerate our workloads by a large amount. On a real machine described in Section 3.1 our workloads take from 5 to 188 seconds to process a single frame. Acceleration by an order of magnitude or more can enable improved accuracy and modeling of new effects. It may even enable some of them to become interactive or real-time applications.

All major microprocessor vendors are now offering chip multiprocessors (CMPs), also called multi-core processors. We expect the number of cores on CMPs to increase steadily for the foresee-

[1]While phenomena at these extreme scales are often depicted in motion pictures, they are typically procedurally and artistically rendered, rather than simulated.

able future, so that CMPs capable of executing applications tens of times faster than today's uniprocessors are on the horizon. Such CMPs would provide exactly the acceleration opportunity needed for production-quality physical simulation applications.

However, for an application to harness the computation power of such a CMP, it must effectively utilize multiple threads. Parallelization of a large code base as used by production-quality physical simulation applications is not trivial, especially when the target parallel scalability is tens of threads.

Our contributions are:

1. We have parallelized three state-of-the-art real-world production-quality physics applications that represent and span the physical simulation problem space: fluid dynamics [4], human face animation [11, 12, 15], and cloth simulation [1, 2]. Our parallelization goal was to achieve performance scaling up to at least 64 threads.

2. We provide a detailed characterization of the key modules of our applications using real-world inputs via simulation of a CMP with 64 cores. Our characterization focuses on the parallel behavior of the applications, including their parallel scalability and synchronization behavior. We also examine their memory behavior, including the working sets and inter-thread communication. Our key findings are:

The modules have parallel scaling on 64 threads of at least 23x. Our modules see significant performance benefits with an increasing number of threads up to at least 64 threads. We expect even higher performance on 128 threads or beyond for all but one module. For the portions of the applications covered by the key modules, we achieve parallel scaling on 64 cores of 45x, 50x, and 30x for fluid, face, and cloth simulations, respectively.

Most modules are parallelized via data-level parallelism. For most modules, the primary data structures are a representation of the physical system being simulated. The modules typically perform their computation by iterating over the data structures. To parallelize these modules, we partition the data structure (i.e., split the loop), with each partition defining a parallel task. Some of these modules are not purely data parallel and require locking along partition boundaries. For others, we replicate data around partition boundaries to avoid locking.

The best serial algorithm does not always lead to the best parallel algorithm. While the initial implementation of a key module in fluid simulation used one algorithm, our best parallelization of the module uses an alternative algorithm. The alternative is 57% slower when serial, but has significantly more thread-level parallelism, and so is 116% faster with 64 threads.

Some modules are dominated by small parallel regions. These regions are entered often enough to account for a large portion of the applications' execution time, so their scalability is important. However, being small makes these modules sensitive to task scheduling overheads and barrier cost.

Load imbalance reduces parallel scaling for many modules. 9 of our 14 modules have more than 10% load imbalance when run with 64 threads. This is due to variance in task size and too few tasks. The modules that suffer from load imbalance would incur significant overheads (e.g., redundant computation) to increase the number of tasks — we believe we have chosen reasonable tradeoffs between load imbalance and parallelization overhead.

Two modules have high parallelization overhead from locking. These modules use locking to protect small operations in a tight loop, leading to high parallelization overhead (>60%). However, the locks have little contention, so this does not impact parallel scalability.

Few modules have high inherent inter-thread communication. While six modules have a large fraction of their L1 misses to shared data, false sharing is the source for two of those, and barrier synchronization is the source for another two. The lack of inter-thread communication is largely intentional, since we a priori assumed it would limit scalability (in many cases we avoid it with redundant computation).

Most modules are streaming. Many of the modules perform numerical operations on vectors, matrices, and/or elements in a spatial representation of the problem (i.e., a grid or mesh). This leads to a streaming access pattern. Some modules repeatedly stream over their data structures (e.g., iterative solvers), leading to a large amount of temporal locality. In those cases, the last-level per-thread working set size also decreases with an increasing number of threads, due to our use of partitioning for parallelization.

Some modules have high on-die and off-die bandwidth usage. Many of the streaming modules also perform relatively little computation per element per invocation or iteration. This results in high bandwidth usage. Five modules have on-die communication-to-computation ratios of at least one byte per ALU operation, and three have off-die ratios exceeding 0.5 bytes per ALU operation. Since CMPs have a large number of threads sharing both the on-die and off-die interconnects, those must provide high bandwidth or this will limit the scalability of these applications.

2. APPLICATIONS

Our applications are derived from the physical simulation package called PhysBAM, developed at Stanford [9]. The package includes techniques for solving a variety of physics-based modeling problems such as fluids, rigid bodies, and deformable solids. The code base has more than 150,000 lines of code. It has not been hand-coded with single-instruction-multiple-data (SIMD) instructions.

We have chosen to study three physical simulation applications that are representative of and span the space of physics-based computer animation: computational fluid dynamics using a particle level-set method, facial simulation using the finite element method, and cloth simulation using a mass-spring system. Each is representative of the state-of-the-art in its respective domain. Figure 1 shows example output frames from the applications, using the real-world inputs that we use in this study. Face and cloth simulation include similar collision detection modules. For cloth simulation, collision detection is a key part of the application. However, for face simulation it is only a minor part of the application, and is significantly simpler computationally than for cloth simulation. Therefore, we disable the collision detection part of face simulation.

The applications we study are thematically similar to other numerical computing and physical simulation codes from popular benchmark suites such as SPLASH-2 [17] and SPEC CPU2006 [13]. However, our target applications are quite different in scope and context from these other codes. This leads to fundamentally different domains, governing equations, algorithms, and computational behavior. The unique requirements of a special effects production environment typically dictate specific choices of theoretical and algorithmic formulations which substantially limit the overlap with other implementations of numerical computing techniques.

We next describe each of our applications.

2.1 Fluid Simulation

Simulated water volumes are key elements in an increasing number of feature films, making fluid simulation (a.k.a., computational fluids dynamics, or CFD) very common in the special effects industry today. Our fluid simulation application uses the incompressible

Fluid simulation Face simulation Cloth simulation

Figure 1: Sample output frames for our applications.

Navier-Stokes equations to describe the evolution of a body of water with a free surface (as opposed to a fluid confined to an airtight container). The equations of flow are discretized on a Cartesian 3D grid using a finite difference formulation. The fluid-air interface is represented by a signed distance function, defined on the same grid and updated using the particle level-set method [4]. The simulation pipeline employs techniques such as the semi-Lagrangian method [14] for velocity advection, the solution of a Poisson equation for making the velocity field divergence free using a preconditioned conjugate gradient (PCG) algorithm, and the use of the fast marching method [10] and fast sweeping method [18] for reinitialization of the signed distance function.

2.2 Face Simulation

Face simulation animates an anatomical model of a human face driven by the action of facial musculature and the motion of the jawbone [11, 15]. A time sequence of muscle activation values and kinematic parameters for the jaw motion is provided as input. The Finite Element Method is used in conjunction with an accurate muscle constitutive model to define forces on a tetrahedral mesh discretization of the flesh. The application assumes that casual facial motion exhibits negligible ballistic or inertial effects. It thus animates the face model as a sequence of *steady states*. Each state is defined solely by muscle activation values and the position of the cranium and jawbone. Such a state is efficiently computed via a quasistatic solver which uses Newton-Raphson iteration to determine the steady state as the solution to a nonlinear system of equations.

2.3 Cloth Simulation

Cloth simulation animates the time evolution of a deformable surface having the material properties of a fabric sheet. The cloth surface deforms under the influence of external forces such as gravity or forced stretching, internal forces such as the elastic response to tensile stress, shearing, and bending, and reacts to collisions with itself or elements of the environment. The deformable cloth is implemented as a mass-spring system where the simulation particles are connected into a triangle mesh. The simulation mesh is endowed with a network of spring elements aligned with all edges and all triangle altitudes. Bending elements provide additional forces that resist change in dihedral angles formed by adjacent triangles.

Each time step of the deformable object evolution consists of a forward time integration followed by collision resolution [1, 2]. We employ a semi-implicit Newmark integration scheme that uses explicit integration for position updates and implicit solvers for velocity updates. A collision resolution algorithm then detects and attempts to resolve collisions of the cloth with itself or other objects that were caused by the integration step. Collision detection is accelerated using a topological hierarchy of bounding boxes constructed every few frames. Should the collision resolution fail, the integration-collision loop is repeated with a reduced time step.

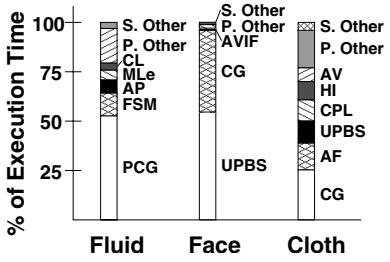

Figure 2: Execution time breakdown for each application.

3. PARALLELIZATION

The applications we study are all computationally demanding — on a real system described in Section 3.1 they take on average 188, 14, and 5 seconds to process a single frame for fluid, face, and cloth simulations, respectively. Since they will all benefit from a large performance boost, we parallelize the applications, targeting a large-scale CMP (i.e., one with tens of cores). Given the size of the code base, we have not parallelized 100% of the applications, but we have come close enough ($\geq$ 96% for all applications — see Section 3.1) to demonstrate that this is achievable. In some cases, our parallelization impacts the output values. This is acceptable as long as the approximations are still visually realistic. Thus, we visually checked the results to verify acceptable output.

We profiled the serial applications on a real machine to select the most expensive modules as targets for parallelization (Section 3.1). We then parallelized those modules, as described in Sections 3.2 to 3.5, targeting a large scale CMP. We discuss only the most expensive subset of the modules we parallelized. After parallelization, we measured the performance of and characterized this subset of modules using simulation since no large scale CMP is currently available (Sections 4 through 7).

3.1 Selecting Representative Modules

We took the conventional approach to parallelizing large code bases: prioritize the modules of each application and parallelize them in decreasing order of importance. Therefore, we first profiled each application to determine the most expensive modules in a serial execution.

For profiling, we used a real 4-way Intel®Xeon™3.0GHz system, with 16GB of DDR2-3200 and 3-levels of cache on each processor: 16KB L1, 1MB L2, and 8MB L3. Maximum memory bandwidth is 6.4GB/s. I/O is minimal for all three applications on this system. For each application, we ran long enough to amortize any warmup effects and also to average out input-dependent behavior.

Figure 2 shows the breakdown of the execution time of each application. We label the key modules (i.e., the most expensive ones) that we will focus on for the rest of this paper. For all three applica-

tions, the key modules make up more than 75% of the application execution time. We also indicate the aggregate time taken by the other parallelized modules with "P. Other" and the remaining aggregate serial modules as "S. Other." The serial modules make up only 3%, 1%, and 4% of the execution time for fluid, face, and cloth simulations, respectively. Further, almost all of the serial modules are easily parallelizable (e.g., array copy operations). The modules that are not easily parallelizable account for less than 0.1% of the execution time. The exception is a kd-tree building module in cloth simulation, which accounts for 2% of the execution time. However, kd-tree building has been shown to be parallelizable [6].

3.2 Parallelization Framework

The applications were parallelized using the fork-join model [5, 8, 16] in which the program consists of alternating serial and parallel sections. This model is attractive because it allows one to start with a serial program and selectively parallelize the most profitable portions of the program until satisfactory performance is achieved.

We use a task queue model to parallelize all modules. A parallel section of the program is specified as a list of tasks that can be executed concurrently. The term *task* denotes a parallel unit of work. In our infrastructure, a task is specified as a function pointer together with its arguments. To scale to N cores, the program has to expose at least N tasks. If the execution time of all the tasks are roughly the same, it might be sufficient to expose exactly N tasks. However, if the task sizes vary significantly, the parallel section has to be broken into enough tasks to avoid load imbalance.

We implemented distributed task queues with work stealing, an effective and widely used task scheduling technique. We optimized the implementation so that it was efficient even for small tasks and a large number of threads. We have benchmarked our implementation against Cilk [5], TBB [16], and OpenMP [8] to make sure that our implementation was efficient.

3.3 Fluid Simulation Modules

Many of the fluid simulation modules operate on a 3D uniform grid representing the space being simulated. To parallelize these modules, we typically partition the grid into cubes, each containing a small number of grid cells. In some cases, the cubes are independent, and in others we require locking to protect reads and writes to grid cells along partition boundaries.

Advect Particles (AP). The particle level-set method utilizes a dual representation of the fluid-air interface in order to improve accuracy and minimize volume loss. Weightless marker particles are seeded on either side of the zero isocontour of the signed distance function corresponding to the fluid-air interface. Those particles are advected along with the fluid and are used to correct the shape of the interface as encoded by the signed distance function. The AP module uses the second order Runge-Kutta integration scheme to advance these particles with their underlying fluid velocities. It iterates over the simulation grid, using the information in a small 3D window around each grid cell to update the positions of the particles it contains. For parallelization, we partition the grid into cubes, and operate on the particles in each cube independently.

Construct Levelset (CL). CL rasterizes rigid bodies that interact with the grid so that their effect on fluid evolution can be computed. CL operates by iterating over the simulation grid and interpolating a signed distance value from the surface of the object using the object's kinematic state and its own signed distance representation in its reference position and orientation. For parallelization, we partition the grid into cubes, and operate on each independently.

Fast Sweeping Method (FSM). The signed distance function receives updates and corrections to ensure that the zero isocontour accurately tracks the fluid-air interface. However, away from the zero level set, the signed distance property may be violated as a result of these corrections. The Fast Sweeping Method [18] is used to re-initialize the level-set values to restore the signed distance property without perturbing the location of the zero isocontour. It has three phases of operation on each cube of the simulation grid: the first initializes each grid cell, the second performs sweeps over each cube in all eight different diagonal directions to propagate interface information, and the final to flag cells far from the interface. For parallelization, we use duplication of cells around the boundaries of the cubes to make the cubes completely independent. Since multiple sweeps may be performed simultaneously on a single cube, we use locking to protect the grid cells.

Modify Levelset Using Escaped Particles (MLe). During the evolution of the fluid volume, the dual representation of the fluid-air interface might become inconsistent; that is, particles that had previously been marked as belonging to the fluid side of the interface might have crossed over to the air region according to the signed distance function, and vice versa. MLe uses the information from the particles in the vicinity of each grid cell to update the signed distance function in each cell. Fine grain locking is employed to guard updates at the boundaries of cubes.

Preconditioned Conjugate Gradient (PCG). Our fluid simulation models an incompressible fluid (i.e., the volume remains constant). To enforce incompressibility, velocity updates must maintain a "divergence free" property. This is accomplished through a projection operation that involves the solution of a Poisson equation. The system matrix is sparse, symmetric, and positive definite, allowing the use of a fast conjugate gradients (CG) solver. Furthermore, PCG uses an Incomplete Cholesky *preconditioner* to substantially accelerate convergence of CG. PCG involves forward and backward substitution, matrix-vector multiplication, and vector dot products. For parallelization, we employ a red-black reordering scheme [7]. This breaks the matrix into a set of red blocks and black blocks, where blocks of the same color are independent of each other. This obviates the need for fine-grained locking at the cost of slightly slower convergence.

3.4 Face Simulation Modules

Parallelization of all modules used in our Finite Element simulation framework is based on a static partitioning of the simulation mesh. Tetrahedra and edges at the boundaries of partitions may span nodes belonging to two or more partitions. In this case a full copy of each such element is given to each partition. This eliminates the need for locking at the expense of repeated computation.

Update Position Based State (UPBS). UPBS uses an iterative Newton-Raphson algorithm to find the steady state of the simulated mesh as the solution to a nonlinear system of equations. In each iteration, this system is approximated by a symmetric and positive definite *linear* system. The UPBS kernel precomputes the matrix of this system. The sparsity structure of this matrix allows its storage in two one-dimensional arrays, respectively indexed by node and edge indices in our mesh. The aggregate matrix is the sum of the contribution of each element (tetrahedron) in our mesh. We circumvent output dependencies by duplicating tetrahedra and edges in the simulation mesh that span nodes assigned to different partitions. We then iterate over the (overlapping) sets of tetrahedra assigned to each partition, computing their contributions to the

global matrix and distributing them to the entries corresponding to the nodes and edges of each tetrahedron.

Add Velocity Independent Forces (AVIF). While UPBS computes the matrix of the linear system arising from the Newton-Raphson algorithm, the AVIF module computes the right hand side of that system. The module iterates over the elements of our simulation mesh, reading the positions of their vertex nodes and computing a force contribution to each of those four nodes. In a fashion similar to UPBS, each partition processes all elements that contain *at least* one node owned by the particle, but only writes the resulting forces on those nodes owned by the partition.

Conjugate Gradient (CG). This module employs the conjugate gradient algorithm to solve the sparse linear system assembled by modules UPBS and AVIF. The fundamental operations performed by this module include a sparse matrix-vector multiplication using the matrix precomputed by UPBS and several streaming operations. As a result of the information duplication by UPBS, the system matrix is encoded in two flat arrays that are accessed in a sequential fashion to perform matrix-vector multiplication. Global reductions necessitate two barriers per iteration of the algorithm.

3.5 Cloth Simulation Modules

Parallelization of some modules in our cloth simulation framework is based on a static partitioning of the simulation mesh, similar to the partitioning of the tetrahedron mesh described in Section 3.4. The set of nodes in our simulation mesh is partitioned between different tasks, and edges and triangles that cross a partition boundary are duplicated between the corresponding partitions.

Update Position Based State (UPBS). Following the explicit position update for the cloth particles at each time step, several position dependent properties such as length and orientation of edge springs or endpoint locations for altitude springs can be precomputed to accelerate subsequent modules requiring force computation. These are all properties of the simplices used in the definition of forces (edges, triangles, and triangle pairs) and their computation is parallelized by simple partitioning of the respective simplex sets.

Conjugate Gradient (CG). The employed iterative Newmark integration scheme calls for the solution of a symmetric, positive definite system to determine the velocity updates for the cloth particles at each time step. The matrix of this linear system is the sum of three matrices, corresponding to the different internal forces (edge springs, altitude springs, and bending elements) considered for cloth simulation. These are sparse matrices, having nonzero entries only for pairs of indices corresponding to particles that are connected by a force, i.e., belonging to the same edge, triangle, or adjacent triangle pair in our cloth mesh. For parallelization, the cloth mesh is statically partitioned, as described earlier. The overall system matrix is not explicitly partitioned; instead each mesh partition implicitly stores possibly duplicated matrix elements sufficient for computing the action of the matrix on the particles it owns. The rest of the CG algorithm consists of streaming operations on the particle velocities, such as vector multiply-and-adds and dot products. Global reductions necessitate three barriers per iteration.

Add Forces (AF). Computation of elastic forces is needed for the explicit parts of the Newmark scheme and as a part of the explicit solver. For parallelization, the mesh is partitioned. For each force defined on a simplex, the positions or velocities of its nodes are used to determine forces on the same particles. Each task writes the forces it computes only for the particles it owns.

Hierarchy Intersection (HI). Collision detection and handling are based on proximity detection of geometrical features of the simulation mesh (i.e., points, edges, and triangles). Proximity queries are performed using bounding box hierarchies. Intersecting two bounding box hierarchies efficiently provides a pruned set of candidate interacting feature pairs from the corresponding feature sets. Algorithmically, this is equivalent to the traversal of a tree with a maximum branching factor of four. This tree has a maximum height of $\log(N)$, where N is the number of features considered. However, the tree is typically sparse, with an expected number of $\Theta(N)$ nodes, out of the $\Theta(N^2)$ maximum nodes allowed by its branching factor. For parallelization, a dynamic partitioning is employed. The tree is traversed in a breadth-first fashion until a given number of independent subtrees has been identified. These subtrees are subsequently traversed in parallel, with their respective results merged at the end.

Create And Prune Lists (CPL). HI returns an unordered list of potentially interacting feature pairs. Before further processing, this list has to be reordered into a list of interactions for each individual feature. Furthermore, geometrically adjacent features (e.g., a triangle and its vertices) will always be registered as potentially interacting, however such results are false positives which can be pruned. For parallelization, the unordered list of feature pairs is partitioned and dispatched to different tasks, which use fine grain locking to register the interaction with each of the two involved features. A second sweep partitions the set of features and proceeds to prune their interaction lists of the false positives.

Adjust Velocity (AV). Following the identification of interacting feature pairs, the particles of the simulation mesh are adjusted in response to the detected collision. Each adjustment amounts to reading the positions and velocities of the involved particles and correcting their velocities to account for collisions. The adjustment is applied in a Gauss-Seidel fashion; thus, any correction on a feature pair will influence the correction performed on subsequently processed pairs. An ordered list of feature pairs is partitioned into contiguous sublists, which are dispatched to different threads. Fine grain locking is used to guard reading and writing to particles that are shared among feature pairs processed by different threads.

4. CHARACTERIZATION METHODOLOGY

Since no large scale CMP is available for us to experiment with, we use cycle-accurate simulation to measure performance of and characterize the parallelized workloads. However, the low speed of cycle-accurate simulation forces us to make some practical choices. Simulating a full multi-frame run, or even one entire frame, of one of our applications is infeasible. Therefore, we simulate the set of most expensive modules (i.e., those described in Section 3). Some modules are too expensive to simulate in entirety; however, all such modules are iterative and have uniform behavior across iterations. Therefore, for modules too expensive to completely simulate, we simulate a representative iteration.

Since we only simulate a single invocation of each module, we take care to pick an invocation from a representative and interesting frame. For fluid simulation, we model a ball falling into a cubic container partially filled with water. We choose a frame soon after the ball hits the surface, and model the space using a 150x100x100

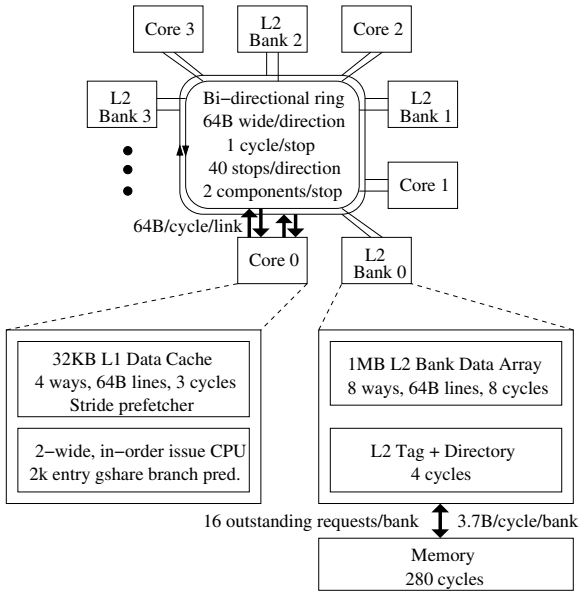

Figure 3: Simulated system.

grid. For face simulation, we model a person speaking a sentence. We choose a frame where the person is in the middle of a word, and model the face with 370K tetrahedra. For cloth simulation, we model a rectangular piece of cloth interacting with a solid sphere. We choose a frame where the cloth is in contact with the sphere, and model the cloth with a 103x61 mesh.

We take care to simulate each of our modules in the context of the full application. That is, we ensure that the input to each module is real and representative of the inputs that the module will receive in a full, multi-frame run. We also take care that the cache state is close to what it would be when the module is run as part of the full application. We fast forward through the full application (i.e., we simulate the architectural state, but not the microarchitectural state) until we reach the module of interest, and then begin our performance simulations. In cases where cache warming is required, we begin performance simulation early enough to ensure that the appropriate data is touched before the module starts. In these cases, we discard the timing and other statistics from the warmup period, keeping only those from the module of interest.

We use a cycle-accurate, execution driven CMP simulator for our experiments. This simulator has been validated against real systems and has been extensively used by our lab. Figure 3 shows our system configuration.[2]

We model a 64-core CMP, where each core is in-order and has a private L1 data cache, and all processors share an L2 cache. Each L1 cache has a hardware stride prefetcher [3]. The prefetcher adapts how far ahead it prefetches — if it detects that it is not fully covering memory access latency, it issues prefetches farther out. The cores are connected with a bi-directional ring. The L2 cache is broken into 16 banks and distributed around the ring. A given cache line can exist in only one L2 bank according to an address hashing function (XORs the most significant bits with the least significant). Inclusion is enforced between the L1s and L2. Coherence between the L1s is maintained via a directory-based MSI protocol. Each L2 cache line also holds the directory information for that line (i.e., state and sharing vector). The ring has 40 stops, each of which has two components connected to it (i.e., core or L2 cache bank). The ring stops have no extra buffering, so messages already on the ring have priority over those entering the ring — messages on the ring are guaranteed to go forward each cycle.

We assume a very high main memory bandwidth so that we do not artificially limit the scalability of the modules. Each L2 bank has 16 MSHRs, and can submit a request to memory every cycle. This gives an aggregate main memory bandwidth of 59 bytes/cycle.[3]

[2]We also ran experiments on a simulator that includes simulation of a full operating system. The parallel scaling results from that simulator are very close to those we present here. However, that simulator does not give as detailed information as the one used here.

5. INSTRUCTION MIX AND EXECUTION TIME BREAKDOWN

Table 1 gives the instruction mix and breakdown of execution time for single-threaded runs of our modules. All three of our applications are floating-point intensive, but some individual modules are dominated by integer computation (FSM, AV, and CPL). FSM spends much of its time manipulating a heap, while AV and CPL have many branches.

The second part of the table shows the instruction throughput in instructions per cycle (IPC), and also the fraction of execution time from the three primary performance bottlenecks: fetch stalls, ALU stalls, and memory stalls. ALU and memory stalls are cycles spent waiting for a result from an ALU or memory instruction, respectively. Many of the ALU instructions in our modules are long latency floating-point ones. Memory instructions also take multiple cycles, even for an L1 hit. Since the core we model is in-order, performance is sensitive to instruction latencies. FSM, AV, and CPL also suffer from a significant fraction of fetch stalls due to hard to predict, data-dependent branches — branch misprediction rates are 20%, 17%, and 20%, respectively.

6. PARALLEL CHARACTERISTICS

6.1 Parallelization Overhead

Parallelization overhead is the difference between the execution time of the original serial version of the code, T_S, and the one thread execution of the parallel version of the code, T_1. More precisely, parallelization overhead is defined as $\frac{T_1 - T_S}{T_S} \times 100\%$. Figure 4 shows the parallelization overhead for each of the modules on a real machine.

Parallelization overhead comes from several sources. First, the code has to be modified to expose the parallel tasks in the program (Section 3.2). The overhead from this is usually fairly small (task queuing overhead for one thread in Table 2). Second, locking needs to be introduced in some modules to ensure mutual exclusion when performing updates to shared data structures. Of course, during a single threaded execution, there is no contention on the locks and the locking overheads are entirely due to the instructions to acquire and release locks. FSM, AV, and CPL incur significant locking overheads (Table 2). In AV, each critical section requires acquiring multiple locks (typically four). The standard technique to avoid deadlocks when acquiring multiple locks is to sort them so that there is a total ordering on lock acquires. The parallelization overhead in AV is primarily due to sorting and locking. Finally, some modules have to perform extra work to exploit parallelism. In FSM, the grid is partitioned into overlapping tiles to allow them to be processed in parallel. The overlapped regions are processed

[3]$Memory\,bandwidth = \frac{\#L2\ banks \times \#L2\ MSHRs/bank \times line\ size}{memory\ latency}$

Modules		Instruction Mix				Execution Time Breakdown			
		Branch	Int	FP	Memory	IPC	Fetch	ALU	Memory
Fluid	**AP**	4%	22%	25%	49%	0.81	3%	24%	32%
	CL	5%	30%	30%	34%	1.00	7%	33%	8%
	FSM	10%	43%	7%	40%	0.85	10%	32%	15%
	MLe	6%	45%	18%	32%	0.59	4%	17%	48%
	PCG	9%	38%	15%	38%	0.61	3%	31%	32%
Face	**AVIF**	10%	27%	24%	39%	0.57	7%	17%	47%
	CG	1%	18%	29%	52%	0.94	1%	22%	24%
	UPBS	8%	34%	23%	35%	0.91	8%	25%	18%
Cloth	**CG**	2%	17%	23%	58%	0.90	1%	30%	23%
	UPBS	4%	19%	29%	48%	0.61	3%	28%	26%
	AF	2%	16%	23%	59%	0.89	1%	27%	25%
	AV	14%	57%	7%	23%	1.09	15%	12%	15%
	CPL	15%	63%	3%	19%	0.82	13%	14%	26%
	HI	11%	35%	11%	43%	0.60	9%	36%	24%

Table 1: Instruction and execution time breakdown for single-thread runs. For execution time, we show the IPC and the percentage of time taken by fetch stalls, ALU stalls, and memory stalls.

Modules		Number of Threads	Execution Time (M Cycles)	Serial Section(s)	Parallel Section(s)				
				Execution Time	Parallelism Type	No. of Tasks†	Load Imbalance	Overheads	
								Task Queue	Locks
Fluid	**AP**	1	1937.66	0.00 %	Nested Loop	3000	–	0.05 %	–
		64	34.19	0.02 %		3000	7.83 %	3.65 %	–
	CL	1	848.11	0.00 %	Nested Loop	4708	–	0.13 %	–
		64	14.98	0.06 %		4708	1.43 %	10.18 %	–
	FSM	1	1799.67	0.02 %	Nested Loop	125, 1000, 125	–	0.01 %	3.12 %
		64	32.58	2.46 %		125, 1000, 125	9.61 %	1.29 %	3.06 %
	MLe	1	942.67	0.00 %	Nested Loop	2×1500	–	0.09 %	0.95 %
		64	17.97	0.17 %		2×1500	12.57 %	6.81 %	1.55 %
	PCG	1	254.47	0.01 %	Loop	4×1, 4×414, 833	–	0.19 %	–
		64	6.19	0.41 %		4×64, 4×414, 833	14.31 %	11.98 %	–
Face	**AVIF**	1	759.51	0.00 %	Loop	1	–	0.00 %	–
		64	16.99	0.17 %		64	20.29 %	0.27 %	–
	CG	1	16904.88	0.06 %	Loop	406×1	–	0.00 %	–
		64	323.52	10.35 %		406×64	3.68 %	3.82 %	–
	UPBS	1	7073.99	0.00 %	Loop	1	–	0.00 %	0.00 %
		64	146.91	0.02 %		64	14.00 %	0.03 %	0.19 %
Cloth	**CG**	1	56.25	0.11 %	Loop	26×1	–	0.03 %	–
		64	2.09	5.09 %		26×64	15.57 %	25.96 %	–
	UPBS	1	22.34	0.15 %	Loop	3×1	–	0.03 %	–
		64	0.62	6.20 %		3×64	10.18 %	12.93 %	–
	AF	1	9.05	0.23 %	Loop	6×1	–	0.04 %	–
		64	0.39	5.47 %		6×64	17.00 %	31.83 %	–
	AV	1	107.02	0.00 %	Loop	1	–	0.01 %	8.80 %
		64	2.44	0.12 %		64	24.74 %	2.02 %	9.96 %
	CPL	1	44.05	0.13 %	Loop	2, 1571	–	0.02 %	11.75 %
		64	0.98	6.32 %		64, 1571	8.75 %	12.11 %	8.69 %
	HI	1	84.13	0.25 %	Tree	1026, 1	–	0.56 %	–
		64	2.77	8.28 %		1026, 64	21.60 %	19.11 %	–

Table 2: Parallel characteristics of the modules. † This column shows the number of tasks in different parallel sections in the module.

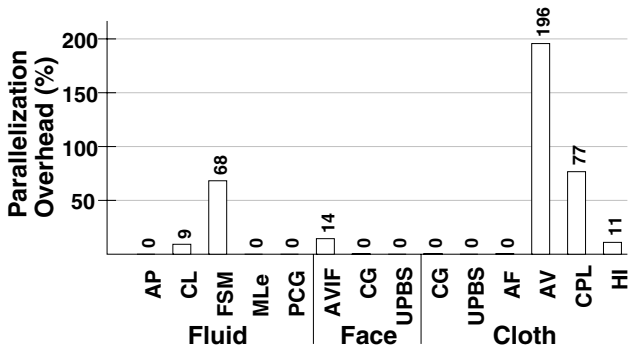

Figure 4: Parallelization overhead for each module shown as the percentage increase in execution time for a single-threaded run of the parallel code versus the serial code. The measurements reported in this table were obtained from an entire application run on a real machine (see Section 3.1). Consequently, the numbers in this table can be compared only qualitatively with the numbers in Table 2. Note that the locking overhead on today's CPUs are significantly higher than what we expect in future CMPs.

multiple times (once on behalf of each tile to which that region belongs). This incurs significant overhead.

For FSM, the original algorithm had bad scaling and was replaced by a completely different algorithm. To do interface propagation in fluids, the original serial code used the Fast Marching Method (FMM) [10]. However, FMM is not scalable to a large number of threads (we see only 20.8x on 64 threads). Consequently, an alternative technique called Fast Sweeping Method [18] was employed. FSM is 30% slower than FMM for one thread but has much better scalability (54.5x on 64 threads).

6.2 Scalability

Figure 5 shows the scalability of the parallel version for all the modules. Most modules show fairly good scaling behavior.

We now look at the reasons why these modules do not deliver linear scaling.

Serial Sections. Amdahl's law dictates that the scalability of a module is bound by the size of its serial sections. For instance, if 1% of the execution time on a one thread run is serial, it limits the scaling of the module to about 39 on 64 threads. Table 2 shows that the size of the serial sections in the various modules is reasonably small. For the one thread runs, the serial code accounts for much less than 1% of execution time for all modules.

Locking. Table 2 shows the overhead of grabbing locks to access shared data in the different modules. In every case, the locking overhead does not increase with the number of threads. This indicates that there is little contention on the locks and that the locking is not a significant factor to the scalability of these modules. This is because the overhead to access an uncontended lock mostly impacts the parallelization overhead (Section 6.1) and not scaling. Note that the locks used to implement the task queuing are accounted for separately as part of task queuing overhead (see below).

Load Imbalance. Table 2 shows the load imbalance on 64 threads for the modules. For some modules, the load imbalance is relatively large, making this one of the primary limiters of parallel scaling.

Modules	2nd level	Fits?	3rd level	Fits?
Fluid	N = grid steps/dimension			
AP	Streaming			
CL	Streaming			
FSM	$\frac{N^3}{P}$	Maybe		
MLe	Streaming			
PCG	$\frac{N^3}{P}$	Maybe	$\frac{N^3}{P}$	Maybe
Face	N = # of tetrahedra			
AVIF	Streaming			
CG	$\frac{N}{P}$	Yes	$\frac{N}{P}$	Maybe
UPBS	Streaming			
Cloth	N = # of triangles C = # of potential collisions (typically on the order of N)			
CG	$\frac{N}{P}$	Yes	$\frac{N}{P}$	Yes
UPBS	Streaming			
AF	$\frac{N}{P}$	Yes		
AV	$\frac{C}{P}$	Yes		
CPL	$\frac{C}{P}$	Yes		
HI	$\frac{C}{P}$	Yes		

Table 3: Per thread working set size growth as a function of input size and number of threads (P), and whether the working set is expected to fit in a reasonable size on-die cache. Working sets that trigger high on-die traffic in our simulated system are highlighted light gray. Working sets that additionally trigger high off-die traffic are highlighted dark gray.

The load imbalance in a parallel section is a function of the variability of the size of the tasks as well as the number of tasks (given in the table). The lower the variability, the fewer tasks are needed to obtain good load balance. Unfortunately, as the number of tasks is increased, the parallelization overhead (Section 6.1) increases. For those modules that perform redundant computation around partition boundaries (see Section 3), the parallelization overhead grows quickly with the number of tasks. For these, we minimize the number of tasks at the cost of significant load imbalance.

Task Queuing. Table 2 shows the overhead of task queuing in the different modules. For some modules, this overhead is large for 64 threads. In our implementation of task queues, all tasks for a parallel region are enqueued before we enter the region (i.e., enqueues are serial code). Therefore, if the number of tasks is large and/or the parallel region is small, the enqueue overhead is large. An alternative implementation of task queues might reduce this and other task queue overheads.

In addition to the reasons listed above, parallel scaling is also affected by the memory behavior. This is covered next.

7. MEMORY BEHAVIOR

We now examine the memory behavior of the key modules. For each module, we characterize the working set sizes, the on-die and off-die bandwidth usage, the effectiveness of prefetching, and the data sharing behavior. Our analysis is inspired by an analysis of the SPLASH-2 benchmark suite [17].

7.1 Working Sets

The cache miss rate versus cache size curve can provide us with insight into how much temporal locality each module has, as well as how effective caches will be at reducing bandwidth usage to the next level of the memory hierarchy.

A knee in such a curve is commonly referred to as a working set size. A working set is a group of data objects with similar temporal locality. Having a cache at least as large as a working set provides

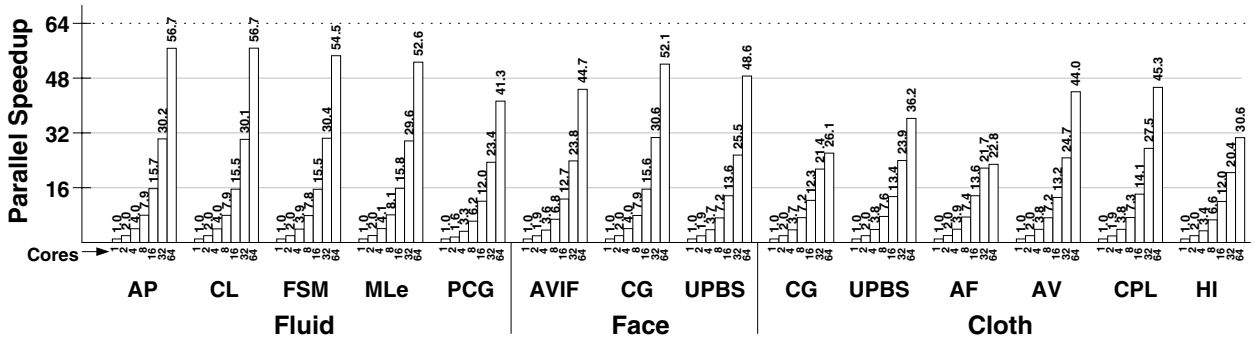

Figure 5: Parallel scaling for each module. Note that the execution time is normalized to the execution time of the parallel version running on one thread. This was done to emphasize the scaling trend of the parallel version. To obtain speedups over the serial version, the data in this graph has to be combined with the data in Figure 4.

a significantly lower miss rate than having a cache just smaller than the working set. Depending on how large the reduction in miss rate is, and the characteristics of the cache hierarchy, this may result in a significant bandwidth reduction and/or a performance boost due to lowered average memory access latency.

Typically, applications have multiple working sets because different data structures in the application have different temporal locality. Also, a parallel application's working set sizes may be a function of the number of threads. For example, an application could keep significant per-thread state that inflates one of its working sets.

Our Inputs. Figure 6 shows cache miss rate versus cache size for our modules for two different scenarios. First, we run each module with a single thread, using a 256MB, 32-way L2 cache. We set the L1 associativity to 32 and vary its size from 16KB to 256MB. Although not shown, all modules except UPBS-face have a first-level working set that is less than 16KB that consists primarily of stack — for UPBS-face it is between 16KB and 32KB. Next, we run each module with 64 threads using the same L1 and L2 configurations, except that we only vary the L1 size from 16KB to 4MB.[4] For both scenarios, we disable the hardware prefetcher since it will mask the working set sizes.

Five modules do not have clearly defined second-level working sets (AP, MLe, AVIF, UPBS-face, and UPBS-cloth). These are largely streaming modules. That is, they touch each data element in their primary data structures a number of times in quick succession and then do not touch it again. The miss rates in the multi-threaded case are somewhat higher for MLe and UPBS-cloth. For the former, the parallelization decreases the spatial locality, and for the latter, the module is small enough that some cold misses to per-thread state drive up the miss rate slightly.

The other nine modules have clearly defined second-level working sets. Most of these modules also stream through their primary data structures, but do so repeatedly. This creates a working set the size of the entire primary data structures for the single-threaded runs. Since most modules partition their primary data structures for parallelization, for multi-threaded runs, most (but not all) have working set sizes of about $\frac{primary\ data\ structures\ size}{\#\ threads}$. FSM, PCG, CG-face, CG-cloth, AF, and CPL all fall into this category.

The three remaining modules, CL, AV, and HI, all have second-level working sets, but not from repeated streaming. For the single-threaded run of CL, the second-level working set is composed of the tasks in the task queue. AV makes a single pass over a list of collisions involving up to four nodes each in the cloth mesh. The second-level working set is the group of nodes involved in multiple collisions. HI performs a tree traversal, so its second-level working set is the entire tree (or subtree for each task, for multi-threaded runs).

Impact of Problem Size. Since bandwidth usage (and latency to a lesser extent) is dependent on whether working sets fit into cache, it is important to understand how the working set sizes can change if the problem size changes. Table 3 shows, for the modules that are not strictly streaming, how the second and, if appropriate, the third level working sets grow with the problem size and number of threads. It also highlights the working sets that trigger high bandwidth usage in our simulated system (discussed further in Section 7.2). The bandwidth usage of the non-highlighted modules is not expected to have a significant dependence on the problem size for reasonable cache sizes. Thus, we focus only on the highlighted entries.

Individual algorithms employed within these applications can in principle be used with various input sizes. However, it is important to note that the end application often limits the range of meaningful problem sizes. Adequate resolution of the physical phenomena being modeled typically mandates a minimum problem size. On the other hand, an arbitrary increase in resolution and complexity of a simulation may render certain algorithmic choices suboptimal, or even call for a different simulation method altogether. The range of reasonable problem sizes is especially strict for face simulation and is most flexible for fluid simulation. We have taken this into account in determining whether a working set will fit into a reasonable on-die cache in Table 3, and also in the following discussion.

PCG's and CG-face's last level working sets for our inputs are larger than our L2; thus, a larger input (higher resolution grid or more tetrahedra, respectively) will not change the bandwidth usage. For PCG, simulating a lower resolution grid may allow the working set to fit in a reasonable on-die cache. We have also simulated PCG with a grid with about an eighth of the resolution of our default input (the smallest reasonable resolution). For this input, the total working set size drops to about 4MB. Thus, for small inputs, the off-die bandwidth usage of PCG may be much smaller than reported here. As mentioned, for CG-face, simulating fewer tetrahedra is not a practical option. Reducing the element count in the model would result in unnatural motion due to under-resolution of the muscle action and create problems for collision detection.

[4] We stop at 4MB because we enforce inclusion between the L1s and L2, and the aggregate capacity of the L1s at 4MB is 256MB, the size of the L2.

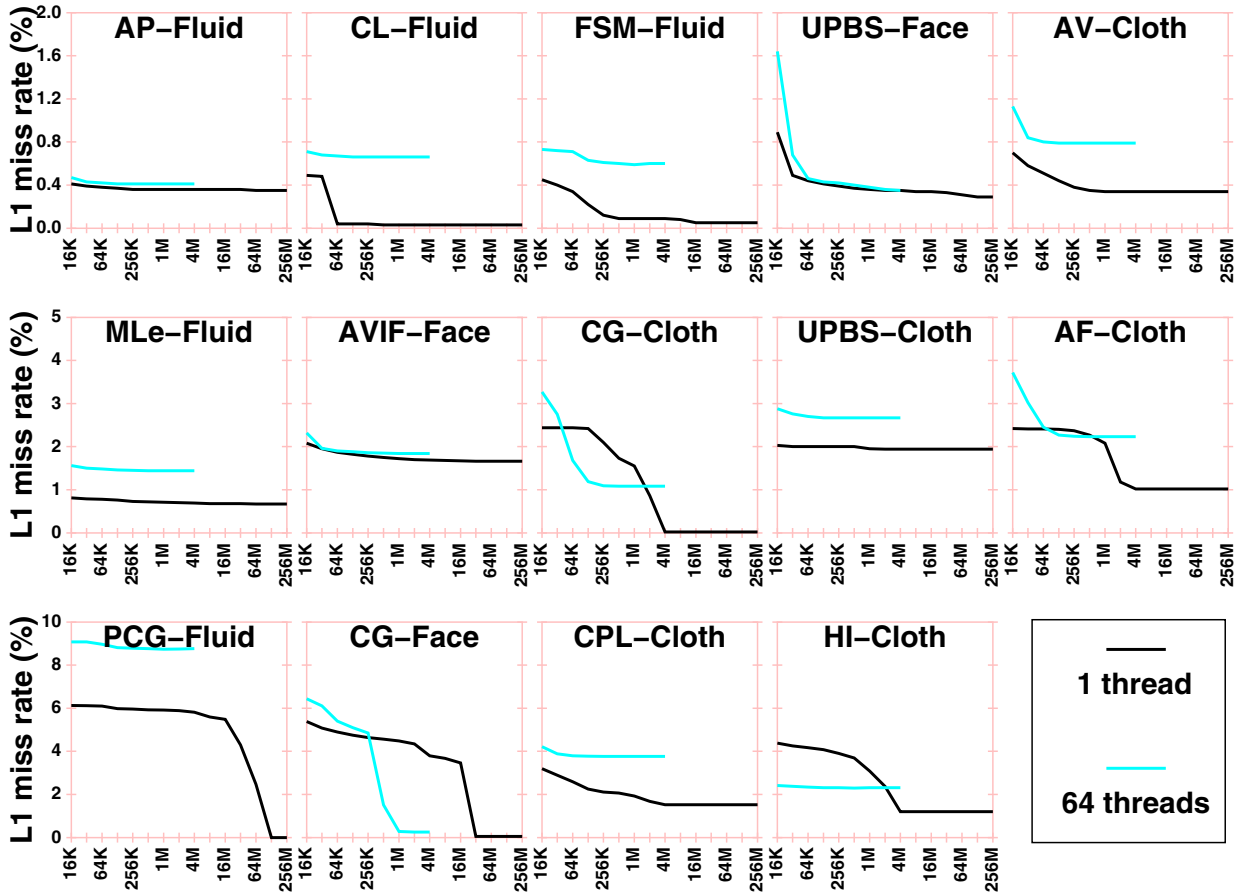

Figure 6: L1 miss rate as a function of cache size. For all experiments, we use a 256MB, 32-way L2, and use a 32-way L1 of varying size. We show results for a single-threaded run, varying the L1 size from 16KB to 256MB, and for a 64-thread run, varying the L1 size for each core from 16KB to 4MB.

For the other modules with high bandwidth usage, we do not expect a change in problem size to significantly alter their off-die bandwidth usage because the working sets should always fit in reasonable sized on-die caches. CG-cloth and HI have relatively low off-die bandwidth usage because of this, although CG-cloth's on-die bandwidth usage is sensitive to the size of the cores' private caches. AF's high off-die bandwidth usage is due to cold misses, and so is not affected by the working set fitting in cache.

7.2 On-Die and Off-Die Traffic

We next examine the amount of on-chip and off-chip data communication. This provides further insights into the memory behavior of our key modules, as well as into the bandwidth requirements of a system that targets our applications. Bandwidth, both on-die and off-die, is a critical resource in scalable CMPs since it must be shared amongst a potentially large number of threads. A module will only continue to scale if sufficient bandwidth is available. We consider only data communication because the number and size of coherence messages are dependent on the coherence protocol and implementation. Data traffic comprises the vast majority of bandwidth usage in our system.

On-die traffic. Figure 7 shows the communication-to-computation ratio for on-die separated into three components: useful prefetches, useless prefetches, and demand accesses (includes writebacks). We include both integer and floating point computation as ALU operations because some modules have significant integer computation.

Most modules' communication-to-computation ratios are relatively insensitive to the number of threads. There are two primary effects that we expect to impact the ratios as the number of threads scales. First, the ratios may decrease as the per-thread working sets shrink. The only module where this effect is pronounced is CL, which matches our expectations from Figure 6. Second, the ratios may increase as inter-thread communication rises with an increasing number of threads. A number of modules see this effect, and we examine it more closely later (see Section 7.3).

Five modules have high on-die bandwidth usage ($\geq$ 1 byte/ALU op) for 64 threads, PCG, CG-face, CG-cloth, AF, and HI. To reduce the on-die bandwidth usage, one might consider increasing the L1 cache size. Our results in Figure 6 indicate that this would help three of the five high-bandwidth modules, CG-face, CG-cloth, and AF. However, increased L1 size would not provide much benefit for PCG or HI because they have high inter-thread communication.

Prefetching. Prefetching into the L1s can also affect the on-die bandwidth usage. Overly aggressive prefetching will increase the bandwidth usage by fetching data that is not used. Further, it may evict useful data that needs to be re-fetched. Figure 7 shows that for

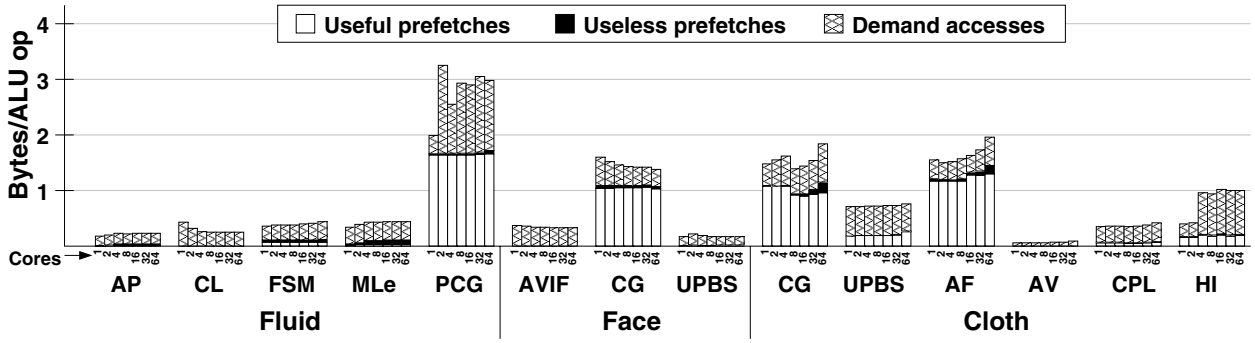

Figure 7: On-die data traffic shown as bytes per ALU operation. The traffic is broken down into usefully prefetched data, uselessly prefetched data, and data from demand accesses (includes writebacks).

all modules, the hardware prefetcher creates little useless prefetch traffic. The figure also shows that for the modules with high bandwidth requirements (except for HI), most of the data communication is from useful prefetches.

Our hardware stride prefetcher is most effective for modules with high spatial locality and predictable access patterns. Therefore, modules that touch large structures in a streaming manner will benefit most. Many of our modules have a streaming access pattern; therefore, most modules with the highest bandwidth usage also see the highest fraction of their data successfully prefetched.

Memory Traffic. Figure 8 shows the data traffic between the shared L2 and main memory, as a communication-to-computation ratio, separated into reads and writes.[5]

The off-die memory traffic for the modules follows similar patterns as for on-die traffic. The off-die communication-to-computation ratios are very insensitive to the number of threads because inter-thread communication is on-die, not off-die, and because there is limited constructive and destructive sharing in these modules.[6]

PCG, CG-face, and AF have high off-die bandwidth usage in addition to the previously discussed high on-die bandwidth usage. On the other hand, CG-cloth has extremely low off-die bandwidth usage because the entire data set for this iterative module fits in the L2 cache. CL and FSM see a similar effect. Also, the off-die bandwidth usage for HI is dramatically lower than the on-die because most of its on-die traffic is related to inter-thread communication.

The off-die bandwidth usage of some of the modules is so high, especially for PCG, that it is likely to limit parallel scalability on most current and near-future systems. On our simulated system, assuming a 3GHz clock, PCG uses an average of 64GB/s of main memory bandwidth for 64 threads. The average bandwidth usage for each of the applications is significantly lower than the peak bandwidth usage. However, the scaling of a worst-case module can limit the scaling of an entire application if insufficient bandwidth is available.

7.3 Data Sharing

Figure 9 shows the on-die data communication-to-computation ratios broken down into four components: non-shared reads from the L2 to an L1, shared reads from the L2 to an L1, cache-to-cache transfers, and writes from an L1 to the L2. We distinguish between shared and non-shared reads from by examining the sharing vector on every L2 access — if another L1 has its sharing bit set, we classify the access as shared, otherwise, non-shared.

Many of the modules are dominated by non-shared reads from the L2 and writes to the L2, indicating little data sharing. This is expected, since most modules partition their primary data structures, and each partition is touched by only one thread. However, six modules have large fractions of their traffic from shared data (CL, FSM, PCG, CG-cloth, AF, and HI). There are three primary sources for the data sharing in these modules.

First, PCG and FSM have true inter-thread communication due to differences in partitioning across parallel sections. That is, these modules contain multiple parallel sections, and do not partition their data consistently across all sections. Therefore, in some sections an element will belong to thread A's partition, while in others it will belong to thread B's. FSM has additional sharing since in its largest section multiple threads may simultaneously operate on a given partition.

Second, CL, FSM, and HI exhibit significant false sharing. CL and FSM update a 3D array partitioned into cubes. Cube boundaries may not be aligned on cache line boundaries, triggering false sharing. HI keeps some per-task state laid out in a 1D array. Updates to this state trigger false sharing.[7]

Third, CG-cloth and AF have very small parallel regions (see Table 2). The barrier cost is significant for these modules when the number of threads is large. Accesses to the shared variables associated with the barriers grows with the number of threads, quickly becoming a large fraction of the on-die traffic. CG-face simulation, UPBS-cloth, and CPL have similar patterns, but they are less pronounced since their parallel regions are larger.

8. CONCLUSIONS

We have studied a set of applications that span the important emerging workload domain of physical simulation for computer animation and visual effects: fluid, face, and cloth simulation. These are all computationally demanding, and therefore can benefit from large speedups. To provide these speedups, we parallelized these applications for a large-scale CMP. We cover code that accounts for at least 96% of the serial execution time for all three applications, and identified that at least 99.9% of the time is parallelizable with reasonable effort. We identified and characterized the most important modules in each application.

[5] Dirty data left in the L2 at the end of each module is not counted as being written back to memory.

[6] Constructive sharing is when a thread brings a line into a shared cache and is subsequently used by another thread; this can greatly reduce off-die traffic. Destructive sharing is when two or more threads are contending for the same set in a cache — this results in additional conflict misses.

[7] This false sharing does not impact performance significantly, so we did not alter the data structure.

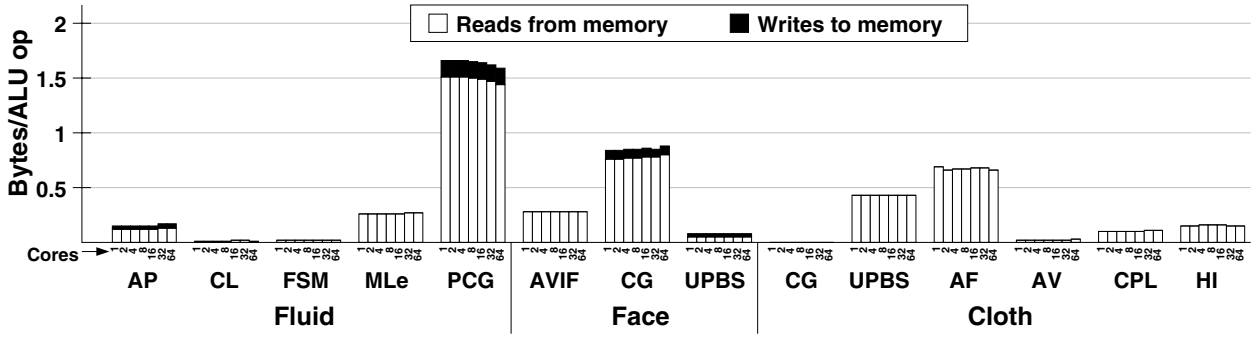

Figure 8: Main memory data bandwidth usage shown as bytes per ALU operation. The traffic is broken into reads and writes.

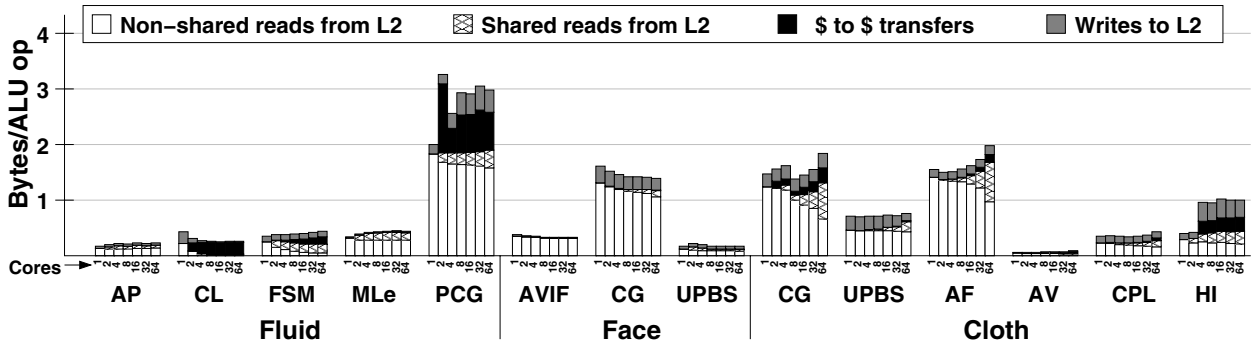

Figure 9: On-die data traffic shown as bytes per ALU operation. The traffic is broken down into reads of non-shared data from the L2 to L1, reads of shared data from the L2 to L1, cache-to-cache transfers (L1 to L1), and writebacks from the L1s to the L2.

For the code representing key modules, we achieve parallel scaling of 45x, 50x, and 30x for fluid, face, and cloth simulations, respectively. The modules have a spectrum of parallel task granularity and locking behavior, and all but one are dominated by loop-level parallelism. Many modules operate on streams of data, sometimes iterating over them, leading to significant temporal locality. This streaming behavior leads to very high on-die and main memory bandwidth requirements. Finally, most modules have little inter-thread communication since they are data-parallel, but a few require heavy communication between data-parallel operations.

Acknowledgments

We would like to thank Bob Liang and Pradeep Dubey for helping to start this effort, Victor Lee and Anthony Nguyen for their assistance with the simulation infrastructure, and Ron Fedkiw for access to PhysBam. We would also like to thank our shepherd, Michael Taylor, and the anonymous reviewers for their helpful feedback.

9. REFERENCES

[1] R. Bridson, R. P. Fedkiw, and J. Anderson. Robust Treatment of Collisions, Contact, and Friction for Cloth Animation. *ACM Transactions on Graphics*, 21(3):594–603, July 2002.

[2] R. Bridson, S. Marino, and R. Fedkiw. Simulation of Clothing With Folds and Wrinkles. In *2003 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 28–36, Aug. 2003.

[3] T.-F. Chen and J.-L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Trans. on Computers*, 44(5):609–623, 1995.

[4] D. P. Enright, S. R. Marschner, and R. P. Fedkiw. Animation and Rendering of Complex Water Surfaces. *ACM Transactions on Graphics*, 21(3):736–744, July 2002.

[5] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 1998.

[6] W. Hunt, W. R. Mark, and G. Stoll. Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic. In *Proc. of the 2006 IEEE Symp. on Interactive Ray Tracing*, 2006.

[7] T. Iwashita and M. Shimasaki. Block Red-Black Ordering Method for Parallel Processing of ICCG Solver. In *Proc. of the 4th Intl. Symp. on High Perf. Computing*, 2002.

[8] *OpenMP Application Program Interface*, May 2005. Version 2.5.

[9] PhysBAM package. http://graphics.stanford.edu/∼fedkiw.

[10] J. A. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, 1999.

[11] E. Sifakis, I. Neverov, and R. Fedkiw. Automatic Determination of Facial Muscle Activations from Sparse Motion Capture Marker Data. *ACM Transactions on Graphics*, 24(3):417–425, Aug. 2005.

[12] E. Sifakis, A. Selle, A. Robinson-Mosher, and R. Fedkiw. Simulating Speech with a Physics-Based Facial Muscle Model. In M.-P. Cani and J. O'Brien, editors, *ACM SIGGRAPH/Eurographics Symp. on Computer Animation (SCA)*, 2006.

[13] SPEC CPU2006. http://www.spec.org/cpu2006/.

[14] J. Stam. Stable Fluids. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 121–128, Aug. 1999.

[15] J. Teran, E. Sifakis, G. Irving, and R. Fedkiw. Robust quasistatic finite elements and flesh simulation. In *2005 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 181–190, July 2005.

[16] *Intel®Thread Building Blocks Reference*, 2006. Version 1.3.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Annual Intl. Symp. on Computer Architecture*, 1995.

[18] H. Zhao. A Fast Sweeping Method for Eikonal Equations. *Mathematics of Computation*, 74:603–627, 2005.